Optimizing Prolog for Small Devices: A Case Study

facultad de informática
universidad politécnica de madrid

M. Carro
J. Morales
Henk L. Muller
G. Puebla
M. Hermenegildo

Authors

M. Carro
mcarro@fi.upm.es
Technical University of Madrid
Boadilla del Monte E-28660, Spain.

J. Morales
jfran@clip.dia.fi.upm.es
Technical University of Madrid
Boadilla del Monte E-28660, Spain.

Henk L. Muller
henkm@cs.bris.ac.uk
C.S. Department
University of Bristol

G. Puebla
mcarro@fi.upm.es
Technical University of Madrid
Boadilla del Monte E-28660, Spain.

M. Hermenegildo
herme@fi.upm.es,herme@unm.edu
Technical University of Madrid
Boadilla del Monte E-28660, Spain, and
Departments of Computer Science and Electrical and Computer
Engineering U. of New Mexico (UNM)

Keywords

Applications of (Constraint) Logic Programming and Prolog, Wearable computers, Program analysis and transformation, Optimizing compilation.

Abstract

In this paper we present the design and implementation of a wearable application in Prolog. The application program is a "sound spatializer." Given an audio signal and real time data from a head-mounted compass, a signal is generated for stereo headphones that will appear to come from a position in space. We describe high-level and low-level optimizations and transformations that have been applied in order to fit this application on the wearable device. The end application operates comfortably in real-time on a wearable computer, and has a memory foot print that remains constant over time enabling it to run on continuous audio streams. Comparison with a version hand-written in C shows that the C version is no more than 20-40% faster; a small price to pay for a high level description.

Contents

1 Introduction

In recent years software has become truly ubiquitous: a large part of the functionality of many devices is now provided by an *embedded* program, which often implements the core tasks that such devices perform. This includes from simple timers in ovens or fuzzy logic based monitoring and control software in household appliances, to sophisticated real-time concurrent systems in cars and cell phones. Upcoming *wearable computing* applications envision an integration of such devices even into clothing (Figure 1).

A range of micro-controllers is available for these purposes which, when compared with the processors currently used in workstations or laptops, are much less expensive and consume a reduced amount of power (starting at micro-Watts for the simplest ones). In return such processors have limited memory (from hundreds of bytes to perhaps a few megabytes total) and speed (up to at most a few hundred megahertz clock rates, and with little or no instruction parallelism). Basically, lower clock rates consume less power and simpler processors with less storage are cheaper.

As a result of this, frequent requirements on *embedded programs* is that they be able to use minimum storage, execute few instructions, and meet strict timing constraints, since all this brings down both cost and power consumption. The importance of requirements depends on the domain. For example, in cars power consumption is less important. Because of these requirements, programs are often developed in low-level languages including, in many cases, directly in assembler [24]. Some of those programs are written on micro-controllers in order to completely minimize power consumption while others are written using also small, but more general-purpose computing platforms [14, 19, 11]. In most cases, platform limitations drive the whole development cycle, diverting attention from modularity, reusability, code maintainability, etc.

At the same time, and despite resource and program development technology constraints, the functionality implemented by embedded systems is often quite sophisticated. This can include, even for the smallest devices, non-trivial matrix operations (as in, e.g., Kalman filters [23], used in GPS receivers), or operating on real-time, intensive data streams (including spatialization, as in the digital sound processing example that we will study in this paper). In addition, more sophisticated functionality and more automated operation is always demanded by users. Furthermore, these systems often face strict correctness requirements because of the nature of the application or simply because of the higher cost of fixing bugs once the system is deployed. In practice, and in order to deal with these conflicting requirements, applications are often coded also in a high-level or specification language, which is used for prototyping and verification, in addition to the above mentioned low-level language, which constitutes the implementation. Unfortunately, often no real link between these two codings of the problem exists.

All this makes the direct use of high-level languages as the actual implementation (and not just as a specification and prototyping tool) an attractive alternative. First, using high-level languages makes it easier to write better programs, with fewer errors, in less time, and with less effort. Problems can be formulated at a higher level of abstraction and much of the low-level detail that must be dealt with when using, e.g., C or assembler (such as manual memory management, ensuring safe typing, complex data structure management, etc.), which complicate and obfuscate the coding of algorithms, are taken care of automatically. These languages also make it easier to detect any remaining bugs and also to verify the correctness of programs automatically. Finally, high-level languages are also useful in the context of the general trend

in processor design towards multi-core chips. Dual processor designs (with four threads total) are present already in mainstream laptops and the expectations are to double the number of cores and threads every two years at fixed cost. Since the motivation behind these multicore designs is precisely to gain performance while keeping resource consumption down, this trend is also likely to hit the micro-controller arena. Parallelized programs will be required to exploit the performance that the chip can deliver, and the parallelization task will add to the burden on the programmer. High-level languages are relevant in this context because they have been shown to be easier to parallelize automatically.

The challenge of course in using high-level languages in embedded and wearable devices is to be able to generate automatically code that is as efficient as required by the platform (i.e., memory, speed, and energy consumption close to the hand-coded low-level implementation). A particular challenge is to achieve this even if numeric or data-intensive computations are involved. While some interesting work has been done regarding the use functional programs in embedded systems, for example work by Peterson et al [16], the use of (constraint) logic programming systems in this context has received comparatively little attention. Logic and constraint programming, and, in particular, the availability in a programming language of logical variables, search, and constraints can be attractive because these features can make it easier to provide sophisticated problem solving, optimization, and reasoning capabilities in devices. This is in line with the demands for higher and more automated functionality from users. The purpose of this paper is to investigate for a particular case study (a sound spatializer embedded in a wearable computer) the feasibility of coding it using a very high level, multiparadigm programming system supporting predicates, logical variables, dynamic typing, search, and constraints in combination with functions, higher order, objects, etc., (in particular, the Ciao system [7, 3, 8]).

However, the point of the study is not to use all these capabilities extensively but instead to study whether current state of the art tools for compile-time analysis, verification, specialization, and low-level optimization are powerful enough to optimize away the default functionality available in such a rich language, including all its libraries, for a program such as the spatializer which only needs a fraction of them. This will require optimizing away all the overhead needed for supporting backtracking, full unification, tagged values, infinite precision arithmetic, etc., which are present by default in the language *for program sections that do not need these features* and see whether it is possible to produce in this way executables for the wearable computer that are competitive in terms of speed, memory consumption, etc., when compared to a solution in a low-level language (in our case, C). This presents challenges that, while having some similarities, are also different for example from those which appear when optimizing functional programs: dealing with logical variables and argument modes (i.e., procedure arguments are not known a priori to be input or output), dealing with backtracking and multiple solutions, eliminating dynamic typing (when compared to strongly typed functional programming systems), etc.

Regarding the case study, it is a stylized (but fully functional) version of a real wearable computing application (designed for the new Bristol CyberJacket of Figure 1) in which a set of virtual sounds are projected into a physical space. The user experiences these *sound-scapes* through a set of headphones attached to the wearable computer (which has limited available power). An example of the use of such a *sound spatialization* device is a "talking museum" where any object, from the actual exhibits to the walls or doors of the rooms, can appear to be talking to the visitor. A compass is fixed on the user's headphones which provides information on head orientation. The wearable computer is also aware of the user's location, through GPS for outdoor locations and through an ultrasonic positioning system [11] for indoor installations. With these two sources of information the wearable device can determine where a sound should

Figure 1: The new Bristol CyberJacket: note the headphones and the glasses (`http://wearables.cs.bris.ac.uk/`).

be positioned relative to the user. By calculating the angle at which the sound is, the delay that the sound will experience at each ear can be calculated, and this allows spatializing the sound [2]. For the sake of simplicity, and since we want to show actual code, we will present a version in which position is not dealt with, and only sound direction is taken into account.

This concrete case study was selected because of its characteristic nature: it requires core functionality present in many wearable computing applications. Handling streams of data such as audio and video and collections of positions is frequent in pervasive and wearable systems. In many common scenarios one or more sensors will produce data streams to be received and used by an actuator. These sensors can generate data at different, unrelated, but generally fixed, and sometimes very high, rates. Therefore, this case study exemplifies a family of programs to which techniques similar to those we will show here can be applied. Very often (including, for example, our case) these problems have, in addition to resource constraints, hard real-time constraints where there are exact deadlines within the system. Of course the objective is to be able to support, in addition to such lower-level data integration tasks, higher-level functionality. But the point of the study is to see if the lower-level tasks can be handled efficiently enough, since the suitability of the programming language used for the higher-level tasks is taken for granted.

The rest of the paper proceeds as follows: in Section 2 the sound spatializer is presented in more detail, as well as the requirements and the characteristics of the platform. Section 3 describes the basic algorithms to be implemented. The actual code, and the results of applying source-level analysis and transformation tools to this code. Section 4 shows the results of applying additional, lower-level optimizations. In Section 5 the performance obtained with the different optimization degrees is summarized and Section 6 presents our conclusions.

2 The Sound Spatializer

The example program that we focus on spatializes sound. Spatialization processes a mono sound stream into a stereo sound stream so that when played through a stereo headphone the sound appears to come from a position in space. The sound stream has to be spatialized in

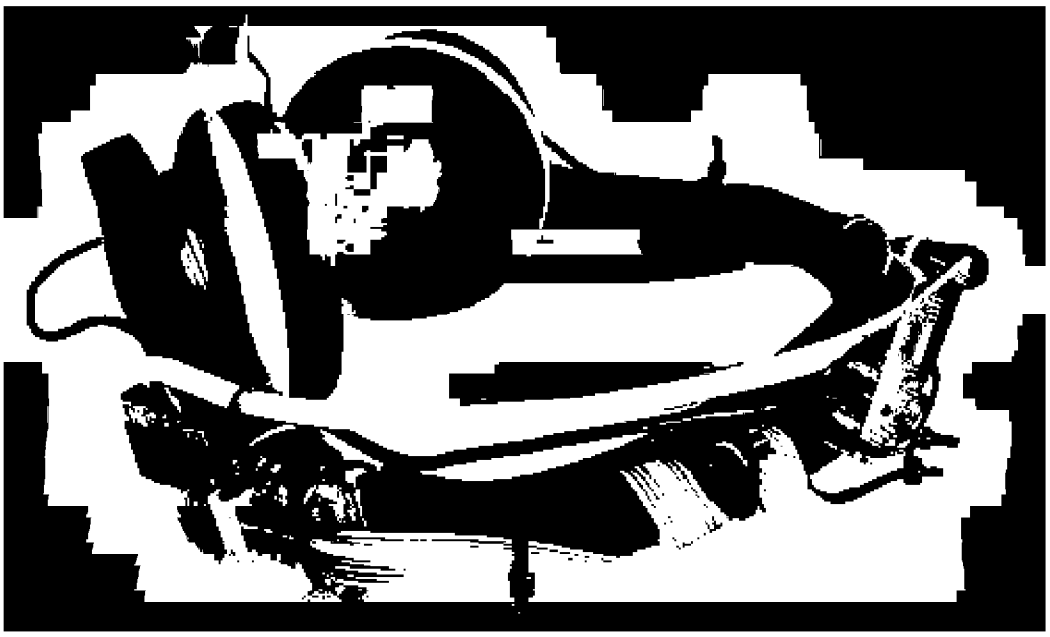

Figure 2: The sound spatializer prototype mounted on headphones. The compass is at the right of the image and the Gumstix board is at the bottom left.

real-time, using heading information from a compass that is mounted on the headphones. When the head is turned, the compass will register a change in heading; the spatialization unit should change the direction of the sound accordingly to create the illusion that it remains fixed at a certain spacial point. Our fully functional prototype has a small processor board (with flash memory), compass, and battery all integrated on a pair of headphones (see Figure 2). The sound stream comprises a series of 16 bit integers. The compass data stream comprises a series of floating point numbers measuring the heading in degrees relative to North. Sound can come from some external source or be stored in flash memory.

2.1 Sound Spatialization Basics

Figure 3 is a simplified sketch of how the ear perceives sound emanating from some point in space. When the head does not face the sound source, sound has to travel a different distance to each ear, resulting in a different time of arrival to each ear. This means that the left and right ear hear the same source with a slight phase shift (in the order of a millisecond), which enables the brain to determine the direction of the sound. In Figure 3, the distances are denoted D_L and D_R, and calculating these distances is the starting point for spatialization. The difference $D_L - D_R$ is a measurement for the phase shift of the signal. The absolute distances can also be used to modify the volume of the sound although in practice attenuation information is hardly used by the brain when determining the source of a sound. D_L and D_R obviously depend on the angle C and the size of the head R.

The computation to be performed on the input data consists of simulating the delay that sound signals will incur when traveling from the (virtual) sound source to the ears. Each earphone is to output one stream of sound samples, which is in turn a possible delayed copy of

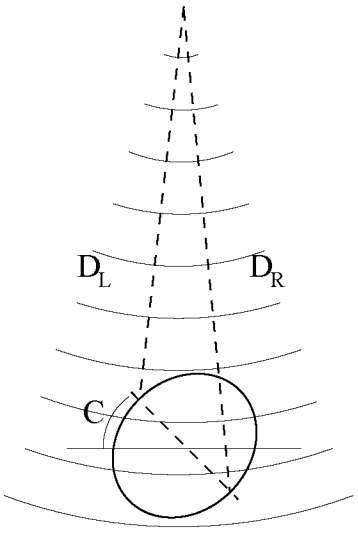

Figure 3: Sound samples traveling from an external source.

the initial sound stream.

A high sampling rate is needed in order to be able to model small head movements. A movement of 3.43 mm is equivalent to a distance of 100 μs; which would need a sampling rate of 100 KHz. The higher the sampling rate, the better the spatialization, but requires more processing. There is a trade-off between quality of spatialization and processing power.

We will assume that signals are delivered at *a priori* known rates, and we will apply analysis and optimization tools in order to reduce the resources needed to deliver sound in a timely manner. The goal is to minimize the execution time required. We want to, at least, be able to execute spatialization in real time on a small 200 MHz XScale processor. Any gains beyond real-time execution will allow us to lower the clock rate of the processor, which reduces power consumption, in turn increasing battery life.

In the following subsections we will discuss the requirements in detail.

2.2 Sound Quality and Spatial Localization

The final application has to produce CD-quality sound, which implies 44,100 16-bit samples per second. With that sampling speed, the minimum relative displacement between ears that can be modeled is about 7.5 mm ($\frac{340\,\mathrm{m/s}}{44{,}100\,\mathrm{sample/s}}$), or 2 degrees rotation. Our program should therefore be able to receive these samples and to deliver 88,200 16-bit integers per second on the two output channels. Concurrently, heading data from the compass has to be read and used in order to produce the sound streams. It is feasible to write a program performing this task in C (see Section 5.3) but the resulting program is inflexible, complex, and it is difficult to check that the code is correct.

2.3 Hardware Characteristics of the Platform

The target architecture is modest in comparison with modern desktops or laptops: it is a Gumstix board,
`http://www.gumstix.com/`, equipped with a 200MHz XScale processor, 64Mb of RAM and 4Mb of flash memory, which acts as permanent storage. It comes with a reduced version of Linux and a cross-compiler which eased the task of porting our Prolog system to the Gumstix. The Gumstix board is around 25 times slower (depending on the application, of course) than a 1.5GHz SpeedStep Centrino, at a fraction of the power usage. The memory and storage limitations are obviously also significant and relevant to our application, since we want to run a non terminating process, and thus garbage collection is critical.

2.4 Hard Real-Time

Ideally a stereo sample should be produced every 22 μs (1/44,100) since this would allow a sound sample to be produced directly on reception of the input. In practice, the sound buffer requires blocks of 256 samples to be written to the sound card, requiring us to produce a block of sound samples every 6 ms. There are two issues that can stop us from meeting this hard real-time deadline:

- Process scheduling may swap out our program for more than 6 ms.
- In the heart of the program run-time libraries, garbage collection (which in principle alleviates the programmer from dealing with several low-level details) could take more than 6 ms.

Both of these problems would result in short breaks in the sound stream, which would be heard as clicks or, in extreme cases, short periods of silence.

From the point of view of the language tools, we have little control over the first problem. In a real scenario one would switch to a form of real-time Linux. In any case, as we will show later, in practice our optimized versions did not suffer from being swapped out.

The second problem above is instead under our control. Fortunately, it turns out that the memory manager of Ciao is good enough as not to cause any noticeable effect on the sound. It also made it possible to run the application without interruptions and with constant, reasonable memory consumption.

In order to give the application more freedom we could increase the buffer size from storing 6 ms. of sound, to storing, say, 600 ms. of sound. However, this would create a lag between moving the compass and hearing the sound move which would render the application unacceptably sluggish and destroy the illusion of spatialization.

2.5 Compass and Concurrency

Accessing the compass requires reading data from a serial interface. Reading the compass data may take an unknown amount of time, because due to limitations of the hardware data may be corrupted on the serial interface. In order not to block the rest of the application, a separate thread is started which asynchronously reads data from the compass, and posts it

```
mono := new InputPeriodicStream(44100);
direction := new InputPeriodicStream(10);
stereo := new OutputPeriodicStream(44100);
while (true) do
        state = f(mono.current(),
                  direction.current(),
                  state);
        stereo.output(state);
end
```

Figure 4: Single-loop algorithm for the spatializer.

```
mono := new InputPeriodicStream(sound_sps);
direction := new InputPeriodicStream(compass_sps);
stereo := new OutputPeriodicStream(sound_sps);
while (true) do
        state := f_c(direction.current(), mono.current(), state);
        samp_sound := sound_sps/compass_sps;
        while (samp_sound > 0) do
                state := f_m(mono.current(), state);
                stereo.output(state);
                samp_sound := samp_sound − 1;
        end
end
```

Figure 5: Pseudo-code for a nested-loop sound spatializer.

for the main program. Communication with the main program is performed via an atomically updatable, concurrent dynamic database [4]. This isolates low-level details of the compass from the rest of the program. However, it makes it necessary for the analysis tools to understand this communication by giving them an appropriate description (via an assertion) of the changes in the dynamic database.

3 Program Code and Source-Level Transformations

3.1 Naive Implementation

The naive implementation of the sound spatialization algorithm is shown in Figure 4. A function f takes the current samples of the sound stream and the direction stream, and produces a stereo sample. We encapsulate knowledge about when to skip samples and any history needed in a separate object "state," making f a pure function. The three stream objects all have preset periodicities The stream objects are initialized with their expected sampling rates.

This code is naive in that inside the function f one needs to perform trigonometric functions on the angle, but these only need to be performed once every compass poll (in our case, once every 4,410 sound samples). In general, a function f that operates on n streams s_0, s_1, s_2, ... can be projected onto a series of functions f_0, f_1, ... such that

$$f(s_0, s_1, s_2...) = f_0(s_0, f_1(s_1, f_2(s_2, ...)))$$

```
spatialize(Samples_Remaining, SampleL, SampleR, CurrSkip):-
     new_sample_cycle(Samples_Remaining, NewCycle,
                      CurrSkip, NewSkip, SampleL, SampleR, NewSampleL, NewSampleR),
     new_sample(NewSampleR, R, RestSampleRight),
     new_sample(NewSampleL, L, RestSampleLeft),
     play_sample(R, L),
     spatialize(NewCycle, RestSampleLeft, RestSampleRight, NewSkip).

new_sample_cycle(0, ~audio_per_compass, CurrSkip, NewSkip, SL, SR, NSL, NSR):-
     skip(~find_skip(~read_compass) - CurrSkip, SL, SR, NSL, NSR).
new_sample_cycle(Cycle, Cycle - 1, Sk, Sk, SL, SR, SL, SR):- Cycle > 0.

new_sample([Sample|Rest], Sample, Rest):- input_sample_if_needed(Sample).

input_sample_if_needed(Sample):- var(Sample) -> read_sample(Sample) ; true.
```

Figure 6: Main loop for the sound spatializer reading from a compass.

If we ensure that s_0 has the highest update rate, and s_{n-1} has the lowest update rate, we can now rewrite the program to a program that computes $f_0(s_0)$ in an inner loop, f_1 in an enclosing loop, ..., and f_{n-1} in the outer loop. For the sound spatializer, the code for this revised program is shown in Figure 5.

We can see that f has been decomposed into f_m and f_c, and that two loops have been created, the outer loop performing f_c at regular intervals, whereas the inner loop applies f_m at a much higher frequency. The code of Figure 5 has lost the simplicity shown in Figure 4, and requires the decomposition of f. Interestingly, the way that f is decomposed depends on the relative periodicities of the streams. If one adds a GPS location stream at 1 Hz, then that will decompose f in a different manner than when one introduces an ultrasonic location system at 30 Hz.

Note that in our example code we only deal with the case where the two frequencies divide each other perfectly. This is not the case for arbitrary sensors.

3.2 High-Level Code for the Sound Spatializer

In order to be able to automate the process of translation, we wrote a complete sound spatializer in Ciao, shown in Figure 6 (see also figures 7 and, later, 8). Similarly to the algorithmic fragments shown earlier, we ignore low level details that deal with obtaining heading data and producing audio data. Figure 6 shows the *actual* control code of the sound spatializer.

The syntax of Ciao Prolog allows the use of functional notation. The prefix operator `~` in Figure 6 enables the use of a predicate as a function by making its last argument correspond to the function result. Hence, one can write the goal `?- append([1], [a], R).` as `?- R = ~append([1], [a]).`

Sound is handled as a list of samples (the samples themselves being of some opaque type), while the compass is explicitly polled (polling is the functionality that the hardware offers). Input samples are kept in an open-ended (incomplete) list which is further instantiated as more samples are needed. Samples which have been read from the sound source but which have not

```
sound_sps   := 44100.     % Samples per second
compass_sps := 10.        % Samples per second
sound_speed := 343.       % Meters
head_radius := 0.1.       % Meters
pi          := 3.141592.

audio_per_compass :=
    integer(~sound_sps / ~compass_sps).

samples_per_meter := ~sound_sps / ~sound_speed.

ear_dif(Angle) :=
    ~head_radius * sin((Angle * ~pi) / 180).

find_skip(Angle) :=
    round(~samples_per_meter * 2 * ~ear_dif(Angle)).
```

Figure 7: Physical model in the sound spatializer.

"reached" the farthest ear are kept in this list.

This list is kept in memory and the unnecessary parts deallocated when samples previously read in are not needed any longer. The predicate `find_skip/2` determines the phase shift between the left and the right ear. The difference (in number of samples) with respect to the previous displacement between the left and right ears is then passed to `skip/6` which returns two new sample lists for the left and right channels. At WAM implementation level these are two pointers to a single sample list.

Figure 7 shows the section of the code that deals with the physical units and calculations. The numbers and functions there represent either physical constants and laws (such as the speed of sound or the amount of space corresponding to every sample, depending on the sampling frequency) or parameters dependent on particular scenarios (such as sampling frequencies or distance from ear to ear). In Ciao Prolog predicates can be defined in functional syntax, by using `:=` instead of `:-`. This assumes that the last argument represents the function result. Arithmetic expressions are also translated (for a full description of the functional syntax see the system manual or [5]).

The code in Figures 6 and 7 can be compiled as is to WAM-based bytecode, and it can deliver spatialized sound with the required quality in a modern desktop or laptop computer, while responding in real time to the direction signals received from a compass. However it falls short in our target platform: playing a 120 second music track takes 115.95 seconds, which would means 96.6% utilisation. As anticipated in Section 2.4, the remaining processor time is not enough to cope with other tasks without introducing noticeable clicks.

To improve this situation we resort to (a) partial evaluation in order to specialize parts of the program and (b) compile-time analysis both to ensure that the program will not raise any run-time exceptions (due to illegal modes, types, etc.) and to perform optimizing compilation to native code (via C) using the information on modes, types, determinism, and non-failure gathered during analysis.

3.3 Compile-Time Checking

The aim of compile-time checking is to guarantee at compile-time that the program will satisfy certain *correctness criteria*. In most programming languages today the correctness criterion is type correctness, and compile-type checking boils down to type checking.

In the case of Prolog, arguments can in principle be input or output without further restrictions. This results in a very flexible programming language, where procedures are *reversible*. However, it is often the case that predefined (system) predicates require their arguments to satisfy certain calling conventions involving both types and modes (instantiation degree). Failing to satisfy such calling conventions is considered an error. Traditional Prolog systems check at run-time such calling conventions and errors are issued if the conventions are violated. In contrast to traditional Prolog systems, in CiaoPP information obtained by static analysis is used to reason about such calling conventions. To this end, the system has an assertion language [18] which allows explicitly and precisely stating calling conventions, i.e., preconditions for predicates. The Ciao system libraries are annotated with assertions indicating pre- and post-conditions for library predicates. Several assertions expressing different pre-conditions and their associated post-conditions can co-exist for procedures which are multi-directional.

Static analysis in CiaoPP is based on abstract interpretation[6], and it is thus guaranteed to provide safe approximations of program behavior. Such safe approximations can be used in order to prove the absence of violations of a set of assertions, and thus the absence of run-time errors.

For example, in the case of our implementation of the stream interpreter, we use the system predicate `is/2`. The arithmetic library in Ciao contains an assertion of the form:

```
:- trust pred is(X,Y) : arithexpression(Y) => num(X).
```

which requires the second argument to `is/2` to be an arithmetic expression (which is a regular type also defined in the arithmetic library) containing no variables, and also provides the information that on success the first argument `is/2` will be instantiated to a number. Analysis information using the *eterms* [22] abstract domain allows CiaoPP to guarantee at compile time that the program satisfies the calling conventions for system predicates (in this case just `is/2`) used in the program. Thus, no run-time errors will be produced during the execution of our code for the stream interpreter.

Also, optionally, the user may provide assertions for his/her own procedures. If available, CiaoPP will try to check at compile time such assertions. Clearly, the more effort the user puts into writing assertions, the more guarantees we have of the program being correct.

3.4 Partially Evaluating the Program

The code in Figure 7, while conceptually clear, performs repeatedly the same set of operations, many of them involving constants. While the part of the main loop dealing with arithmetic is not called a large number of times (because of the low sampling rate of the compass), certainly opportunities exist for partial evaluation to improve execution time. Indeed, the Ciao partial evaluator reduces all the code in Figure 7 to a single clause:

```
find_skip(A,B) :-
```

```
C is sin(A*0.017453288889),
B is round(25.94117647058824*C) .
```

All predicates and facts which calculate numerical values in order to implement the physical calculations are evaluated at compile-time and the results propagated to the appropriate places in the program. Other parts of the program (such as loops, etc.) are also specialized, but the effect in those program points is less relevant. The tools can handle input/output and other built-ins, since they are appropriately annotated with assertions in the libraries where they are defined.

Partial evaluation by itself gave, on average, speedups ranging from a factor of 1.15 to 1.2 on an i686 and around a factor of 1.1 on a Gumstix, when the compass is polled at 10Hz (see Table 1). On the Gumstix, partial evaluation decreases the processor utilization to 86.2% —substantially better than with the non-specialized code. Speedups are even higher when the compass data is read more frequently. As shown in Table 2, the speedup in this case (still using only partial evaluation) on an i686 is between 1.69 and 1.92 when the compass is polled for every sound sample.

Although these results are encouraging, specialization by itself still does not increase performance to a level where the spatializer really runs reliably in real-time on our target platform. Since no further source-to-source transformations will improve performance significantly, the next step in order to optimize the program further is to compile it into native code, using information from compile-time in the process. As we show in the following section, this finally results in an executable that meets the timing and memory requirements, while still starting from a high-level program with no user-provided information for types, modes, etc.

4 Optimized Native Code

Two orthogonal issues affect the performance of the sound spatializer: the time taken to process each sample, regardless of how it is processed, and the time taken to compute the new delay to be applied to the output streams. The former concerns mainly data-structure and control compilation (how the main loop is mapped into the lower-level language, how data structures are handled, and how data is read from and written to the streams). The latter is dominated fundamentally by costly (at least from the point of view of Prolog) floating point arithmetic.

We attacked these problems by first compiling to a lower-level language (to native code, via C) first with little use of information not already present in the original program, in order to verify to which extent a straightforward translation will improve execution, and later by adding extensive compile-time information gathered though global analysis.

4.1 Naive Compilation to Native Code

Compilation to native code proceeded along the lines of [13], but without using any information not explicitly present in the original source code (i.e., no information about types, modes, determinism, non-failure, etc.). This process improves performance mainly by reducing the time used in instruction fetching within the main emulator loop and probably also by better data locality in the cache memory. Any data structures created by the Prolog program remain

```
:- true pred new_sample_cycle(A,B,C,D,E,F,G,H)
    :  ( int(A), term(B), int(C), term(D), term(E), term(F), rt2(G), rt2(H) )
    => ( int(A),  int(B), int(C),  int(D),  rt2(E),  rt2(F), rt2(G), rt2(H) )
    +  ( is_det, mut_exclusive ).

new_sample_cycle(0,4410,C,D,E,F,G,H) :-
        read_compass(I),
        find_skip(I,D),
        J is D-C,
        skip(J,E,F,G,H).
new_sample_cycle(A,B,C,C,E,F,E,F) :-
        A>0,
        B is A-1.

:- true pred find_skip(A,B) : ( flt(A), term(B) ) => ( flt(A), int(B) ).

:- true pred find_skip(A,B) : ( mshare([[B]]), var(B), ground([A]) ) => ground([A,B]).

:- true pred find_skip(A,B)
    :  ( mshare([[B]]), var(B), ground([A]), flt(A), term(B) )
    => ( mshare([[B]]), var(B), ground([A]), flt(A), term(B) )
    +  ( is_det, mut_exclusive ).

find_skip(A,B) :-
        C is sin(A*0.017453288889),
        B is round(25.94117647058824*C) .

:- regtype rt2/1.

rt2([A|B]) :- term(A), term(B) .
```

Figure 8: Part of the information inferred for the compass program.

the same, as well as the memory usage, existence (or not) of choice points, etc.

This relatively simple compilation strategy can provide some additional performance. In our case this was actually a turning point for the practicality of the application: the processor utilization in the Gumstix decreased to 81.7% when compiling the non-specialized program and to 73.6% with the partially evaluated program. The performance of the former executable is still not enough to give a smooth playback. However, the latter is fast enough to play and to poll the compass at an adequate pace while supporting some additional, light load in the host processor. It is however still not a totally satisfactory solution, as it is far too easy to produce noticeable interruptions in the playback.

4.2 Types, Modes, Determinism, Non-Failure

One of the tasks that a non statically typed language has to perform at runtime is precisely checking types and, for a logic-based language, also modes. Note that, unlike in other declarative languages such as Mercury [20], Haskell [9], or ML [12] and their variants, in Ciao the initial

program does not need to include any type, mode, determinism, or non-failure declarations.[1] Removing any unnecessary dynamic checks and data conversions should bring about an increase in performance. Some other fundamental areas in which static analysis information can be used to optimize native code generation are:

- Mode and type information can be used to reduce the overhead involved in parameter passing and unification by, e.g., compiling the latter into simple low-level assignments, perhaps with trailing.
- Data can be represented in more efficient ways.
- Determinism and non-failure information make it possible to anticipate that no backtracking will be performed, so the overhead associated to the creation and management of choice points can be reduced or eliminated altogether.

4.3 Automatic Static Inference

CiaoPP is able to infer automatically a significant amount of information, provided that the *boundaries* of the program are well defined. In general, in a module-at-a-time analysis and compilation setting declarations about the exported predicates are provided and they are used as a starting point to analyze the program. When, as is the usual case, sources for all the application code (libraries included) are available and global analysis is performed (maybe incrementally) for all modules of an application (as in our case), then most information can be inferred automatically.

Also, when there is communication with the outside world and the type of incoming data is relevant then this data obviously also has to be described via assertions. In our case the only external data we need to deal with is that coming from the compass, since the sound samples themselves can be treated as having an opaque type.[2] Data coming from the compass is always a floating point number. To reflect this, we added the following assertion for the `read_compass/1` predicate (which accesses compass data):

```
:- trust pred read_compass(X) : var(X) => flt(X).
```

to the module encapsulating the compass access. This assertion should be read as: "in any call to `read_compass/1`, the predicate argument should be free when calling the predicate and it will be instantiated to a floating point number upon call success." No other information is needed by CiaoPP to infer accurate information regarding all the types, modes, and determinism of the program.[3]

As an example of the type of information which is inferred automatically, Figure 8 shows two predicates (`find_skip/2` and `new_sample_cycle/8`) with the corresponding assertions on types, modes, determinism, and sharing information, as they appear in CiaoPP's output.[4]

[1] Note also that mode and determinism annotations are not needed in functional languages because they are implicit in the language design: all functions produce a single solution and all function arguments are input.

[2] If we were to change, e.g., their amplitude, then we would have to do something similar with this data.

[3] On the other hand, if no information about the `read_compass/1` predicate is provided little useful information can be inferred and most of the improvements that will be described in the following sections cannot be achieved.

[4] Much more information on sharing and freeness was produced for `new_sample_cycle/8`. We are however omitting it since it is not instrumental for our case. On the other hand, it would be vital if we were trying to parallelize the code automatically.

Predicate `new_sample_cycle/8` is the only one that is written in Prolog and does not appear in the code in figures 6 and 7. Of the remaining predicates `read_sample/2` and `play_sample/2` are direct interfaces to lower-level libraries accessing device drivers and `read_compass/1`, as we discussed previously, performs communication with the concurrent process that reads the compass.

Predicate `new_sample_cycle/8` is inferred to be deterministic and the clauses are found to be mutually exclusive. This means that a more efficient compilation scheme which does not even consider pushing choice points can be used.

In `new_sample_cycle/8` the angle read by `read_compass/1` is used in the call to `find_skip/2` and the type of the variable returned by `read_compass/1` is propagated. In turn, the second parameter of `find_skip/2` is automatically inferred to be an integer on exit. This is expressed by the first `find_skip/2` assertion which states that in any successful call where the first argument is a float and the second is any term, the second argument will be instantiated to an integer on exit. The third assertion for `find_skip/2` establishes that under certain conditions calls to `find_skip/2` will return a single solution and do not perform any backtracking.

Additionally, the open-ended list used to hold the samples to output is approximated with the type `rt2/1`, which only states that the argument is a *cons* cell. This information, albeit not complete, is enough for a lower-level compiler to generate better code which avoids testing at runtime the type of a parameter.

4.4 Optimizing Tests and Type Conversions

The performance results in Section 4.1 can be improved upon by having the low-level Prolog compiler use the information inferred by global analysis along the lines of [13]. In particular, using just the inferred determinism and non-failure information the processor busy time is reduced to 77% (for the non-specialized program) and 69.8% of the time for the specialized version. The non-specialized program compiled to native code in this way is now able to generate the sound stream and poll the compass simultaneously with quite acceptable sound. If mode (at predicate entry and exit) and type information are also taken used, performance increases and the processor utilizations get further reduced to 73.6% and 66.2% for the non-specialized and specialized programs, respectively.

4.5 Optimizing Arithmetic Operations

The strategy for compilation to native code used so far (i.e., essentially [13]) obtains speedups by optimizing control and eliminating many type and mode tests and conversions. However it preserves the original data representation of the WAM: data is still stored in tagged words,

In most cases, maintaining the data "boxed" inside tagged words does not incur a big performance penalty, since the code generated by a C compiler to do the tagging/untagging is in general relatively cheap in comparison with what is done with the data itself. However, this penalty is comparatively large for operations which are simple enough to be implemented directly in hardware by a single assembler instruction. A case which stands out is that of arithmetic operations which, for floating point arithmetic, suffer from an additional overhead: FP numbers are not carried around directly in a tagged word; rather, the tagged word points to some heap structure which holds the FP number. Therefore, boxing and unboxing an FP

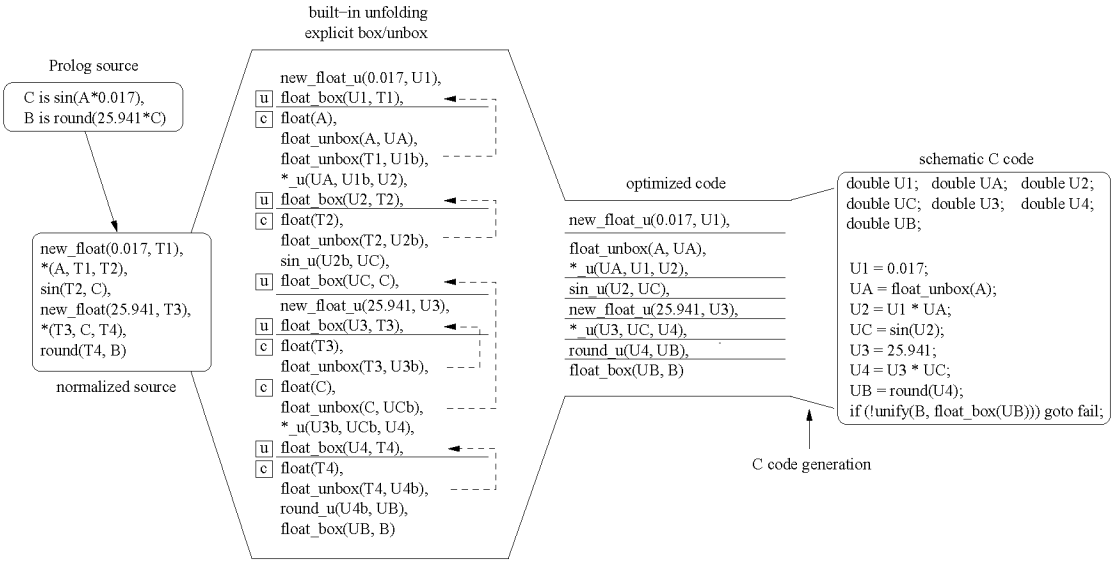

Figure 9: Unboxing optimization.

number is a costly operation and, in principle, this unboxing/boxing cycle has to be repeated every time an FP operation is performed. Keeping FP numbers in the heap needs, additionally, more memory, and garbage collection has to be called more often.

Another big disadvantage of keeping numerical values in boxed form is that when compiling via C the C compiler does not *see* native machine data. This makes it difficult for it to apply many useful optimizations (instruction reordering, use of machine registers, inlining, etc.) which have been devised for more standard C programs.

Unboxing has been studied in functional programming [15, 10] and shown to achieve very good speedups. This is helped in part by the use of strict type systems and the lack of different instantiation modes. Strict typing applies also to the case for Mercury, which does implement boxing and unboxing within this context. Also, information about modes and determinism is compulsory. An interesting related approach, is that of [17] for Haskell. The kernel language was augmented to represent unboxed values and the simplifications to remove redundant operations were formalized as program transformations. However, language differences and the issues on which that work focuses (strictness, polymorphism, etc.) makes a direct application of these techniques difficult in our case.

Unboxing for logic programming systems which are untyped and dynamically tagged has received comparatively little attention. For example, Aquarius [21] did not perform boxing/unboxing, and mainstream logic programming systems, such as, e.g., SICStus, do not perform it when compiling to native code. Perhaps the closest work is that of [1] which proposes a compilation strategy for the concurrent, committed-choice logic language Janus which, starting from a program annotated with type and mode declarations, performs an inter-procedural analysis to determine the best representation of each procedure argument and later an intra-procedural analysis to avoid redundant boxing/unboxing operations.

Our approach to boxing/unboxing in fact shares some ideas with [1], although the languages are quite different. Similar type and mode annotations are required, even if in our case they

Compilation mode	Non-Specialized			Specialized		
	i686	**GumStix**		**i686**	**GumStix**	
	secs.	secs.	Utilization	secs.	secs.	Utilization
Bytecode	**4.70**	115.95	96.6%	**3.91**	103.49	86.2%
Compiling to C	**3.87**	98.08	81.7%	**3.36**	**88.27**	73.6%
Id. + semidet	**3.28**	**92.42**	77.0%	**2.85**	**83.74**	69.8%
Id. + mode/type annotation	**3.00**	**88.38**	73.6%	**2.57**	**79.42**	66.2%
Id. + arithmetic	**2.90**	**85.70**	71.4%	**2.47**	**78.01**	65.0%

Table 1: Speed results and processor utilization for a benchmark with different compilation regimes.

are inferred automatically. However, in our case determinism and non-failure information is also needed and inferred. This is unnecessary for Janus since it does not support backtracking or failure. A similarity with [17] is that we have formulated the problem as a source-to-source transformation on an extended language which includes boxing and unboxing operations. The preliminary implementation used in our experiments only supports unboxed representations for some basic, native types and for temporal variables with a restricted lifetime, in order to ensure that there will be no interaction with the garbage collection.

Boxing/unboxing removal works at source level by exposing the code of builtins which handle word tags. They typically share a similar structure: check types of input arguments, obtain unboxed input values, operate on them, box output values, and unify with output arguments. Informally, the process we use to detect and remove unneeded format changes is:

1. Unfold builtin definitions so that type checking, unboxing and boxing become visible.
2. Make a forward pass to remove redundant unboxing operations. An abstract state is kept which relates boxed program variables with their unboxed version, and it is updated with each unbox operation and consulted when a variable is needed to check if there is an unboxed version available.
3. Make a backward pass to remove unnecessary box operations whose result is not necessary any longer.

The boxing/unboxing state can be updated as clause goals are traversed, by adding pairs of variables and by removing the link between them when they become out-of-sync or when some temporal variable becomes out of scope.

Figure 9 sketches how the algorithm behaves for a short piece of code corresponding to the body of `find_skip/2` (Figure 8). The initial code is shown in the leftmost topmost box. The following box is the result of splitting arithmetic expressions into basic operations (which still work on boxed data) and adding number-creation primitives. Each of these primitives is later expanded into smaller components which either create or disassemble boxed values or work directly with unboxed numbers.

Backward dashed lines with arrows link pairs of unbox / box operations which forward examination is able to simplify, since boxed versions of some variables are not used between them. Goals marked with `C` denote checks (coming from builtin expansion) which are statically known to be true at runtime, either because of assertions at Prolog level or thanks to information gathered in the fragment of code being compiled. They can be safely removed. Finally, goals

marked with [u] are marked as unnecessary during the backward pass because their output value is not used.

The following two boxes show the intermediate program after removal of dead code and, finally, the corresponding C code. Only two operations are used (one box and one unbox) and intermediate variables have been mapped to C (native) variables. This has the added advantage that it makes it possible for the C compiler to apply the usual code optimization techniques.

We applied this unboxing optimization both to the non-specialized and to the specialized Prolog program. In either case, analysis is required to optimize the final code, as type information is needed to detect the places where boxing can be performed. As an example, the information inferred for `find_skip/2` makes it possible to perform all the arithmetic in the predicate using native floating point numbers, since it is known that the input argument `A` is a floating point number. `A` can be unboxed at the beginning and all `float/1` checks can be removed. All intermediate numerical values can also be kept in their native format. Program point information (not shown in Figure 8, for brevity) also shows that the analysis infers the same type for `C`. Additionally, we know that in every call to this predicate the parameter `B` is always a free variable. Therefore, the second call to `is/2` does not need to use full unification, and it is compiled as a simple assignment.

Similar optimizations in other parts of the program lead to performance gains: the unboxing optimization made it possible to reduce processor utilization to 71.4% for the non-specialized program and to 65% when running the specialized one. In both cases this is enough for the Gumstix to respond adequately to compass movements, even if there are several other (non CPU-bound) processes running on it.

5 Summary of the Experiments

Although some of the results obtained were already presented in the previous sections, we now summarize our experiments and results. As mentioned before, experiments were conducted both with the original program and with the automatically specialized version. Both versions were evaluated on a machine with a SpeedStep Centrino running at 1.4GHz and on the Gumstix. The actual test consists of playing a two-minute track[5] while compass data is sampled 10 times per second (a second battery of tests was also performed playing the same track but reading the compass data for every sound sample, which will be discussed later). While the input stream is not infinite, after a few seconds both CPU usage and memory consumption stabilize, which makes us confident that the program would be able to run indefinitely.

Evaluation is based on measuring the total processing time required. In addition, we recorded whether we perceived any artifacts such as clicks and silences. The measured execution time tells us whether we can sustain the bandwidth. Noting the presence of artifacts catches issues such as garbage collection taking longer than the hardware buffer time. The results are presented in Table 1 where scenarios which gave acceptable sound are marked in boldface in. A rough classification of the experiments performed and the processor occupation, and a pictorial summary of their characteristics, is shown in Figure 10.

[5] *Dies Irae*, from Mozart's Requiem.

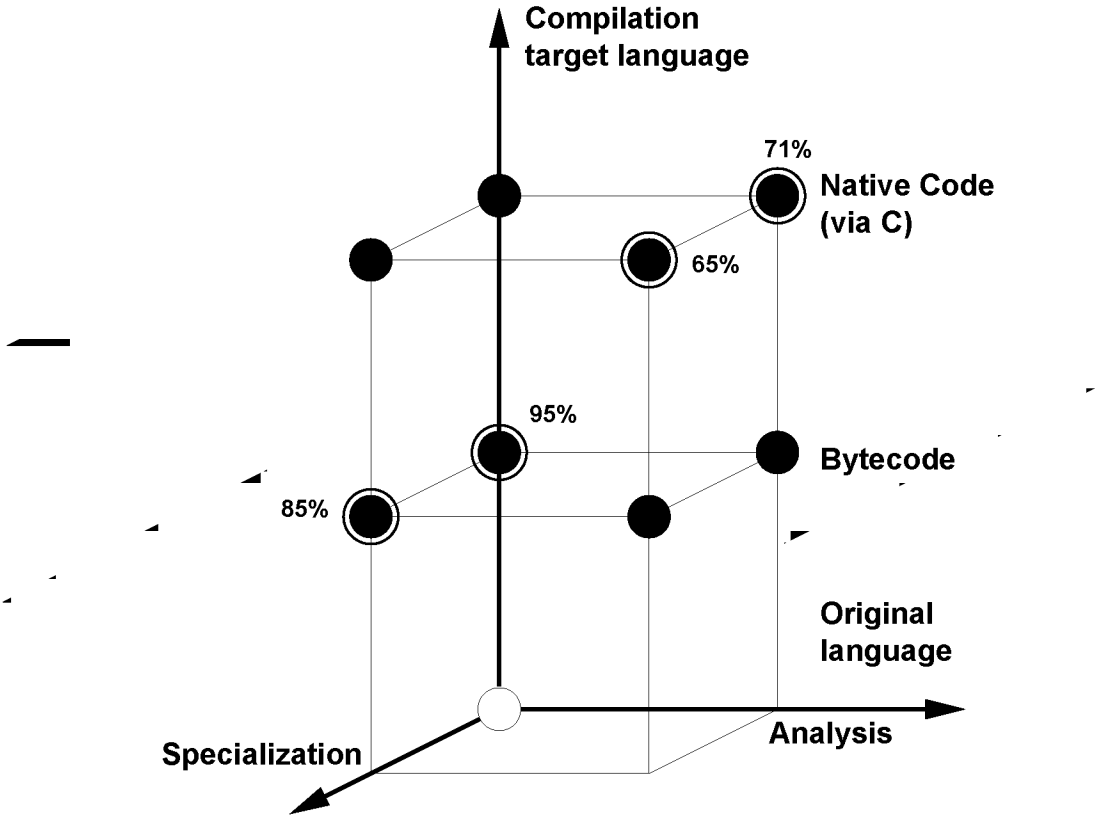

Figure 10: Global view of the experiments.

Compilation mode	Non-Specialized	Specialized
Bytecode	25.64	14.00
Compiling to C	21.59	11.99
Id. + semidet	19.59	11.53
Id. + mode/type annotation	19.19	11.08
Id. + arithmetic	6.97	3.62

Table 2: Results for a series of compilation techniques with a higher compass data rate.

5.1 Basic Results

From Table 1 we can see that the original program (Bytecode, Non-Specialized) is efficient, especially taking into account that it is written in a style that is very close to a specification. But there is not enough spare time to produce a sustained high quality sound stream. A combination of specialization plus compilation, or compilation to C plus some analysis data, is enough to make the program deliver sound acceptably. However, the CPU usage in this case is in the Gumstix close to 100%, and any other process or activity in the same board causes audible interferences. It is only when specialization plus analysis is used to compile to C that the processing speed is enough to support other processes concurrently in the same board without audible interferences.

Regarding the performance improvement obtained, the fastest program in the Gumstix runs 1.5 times faster than the original, slower one. In the i686 machine the difference is larger: the corresponding speedup is around 1.9. However, as we will see in the following section, this is a base value which increases with the frequency at which the compass is read.

5.2 Incrementing the Sampling Frequency

Indeed, the optimizations that we perform impact mainly a fragment of code (the one which computes the phase shift between the two ears) that needs to be executed infrequently with the current compass hardware. Faster updates on the compass hardware (for example, using location information that comes at a rate of 30 Hz or, in another scenario, when rotation accuracy may have to be higher), would require a larger fraction of the processing time to be spent on computing the heading data.

As an experiment, we have also measured the case where we provide the heading data at a rate of 44,100 Hz, i.e., at the same frequency as the audio data. Note that this is the highest polling rate which makes sense, since trying to poll faster than the audio sampling frequency means that compass polls can actually be discarded until the next audio sample is available. Table 2 summarizes the results under that assumption for an i686, where the compass was simulated via software. With these parameters we measured a 7-fold speedup between the slowest and the fastest executable.

This is indeed a very good result, and an extrapolation to the Gumstix (using relations coming from data in Table 1 and assuming a linear model), suggests that with our current technology the software running on the Gumstix would be very close to supporting compass sampling at 22,050 Hz.

The improvement brought about in this case by the introduction of specific floating-point operations and by the use of a specialized version is much higher than in the previous set of

tests. The reason for both improvements is that more time is spent on arithmetic operations in relation with the rest of the program. Therefore, compile-time specialization, which evaluates many FP operations at compile time, is able to simplify fragments of code whose execution would take a substantial portion of the execution time. This is easy to see from the difference between the left and right columns. Something similar happens with the low-level optimization of floating point arithmetic: in this case operations are not removed, but instead they become much cheaper. The effect of this optimization can be seen from the big difference between the last and next-to-last rows.

5.3 Comparison with C

In order to determine how far we are from an implementation in a lower-level language, we compared our performance results with those of a C program. The C program mimics to some extent the Prolog program in the sense that it offers the same flexibility, i.e., buffers are dynamic and their size does not need to be statically determined. However, it does not incur any unnecessary overheads (it was written by an experienced C programmer). The results are highly encouraging, as we found the C program to be only between 20% (for the tests in Table 2) to 40% faster (for the tests in Table 1) on an i686 processor. Interestingly, this C program did not behave as smoothly as the Prolog one when executed in the Gumstix: the many manual memory allocations and deallocations caused audible clicks in the sound output. Also, the complexity of the C program would have made tuning the application much more difficult, and in the end it would probably need the development of an *ad-hoc* memory manager.

6 Conclusions and Further Work

In this case study we have developed a fully functional application (a sound spatializer, intended to run on a wearable computer –the Bristol "CyberJacket") in a (C)LP language. The application had to meet some requirements regarding timing and non-functional behavior. The resulting application code was judged by the jacket developers to be clear and compact, but the initial executions, using a bytecode interpreter in the wearable computer, did not meet the stated requirements. A series of analysis, specialization and optimizing compilation stages were necessary to make it run flawlessly on the target machine. Specialization and compilation was done using an integrated programming environment (Ciao/CiaoPP) which was able to produce an efficient executable from initial high-level code.

In this test case we have shown how a combination of a high-level, very dynamic, logic-based language plus a set of advanced analysis, transformation, and compilation tools can produce a program which deals with a combination of numerical and symbolic processing (in the form of data structures) and executes adequately (in terms of time, memory, and feedback to the user) on a pervasive computing platform, while meeting the real-time requirements imposed.

We have tested the benchmark in two scenarios which differ in the amount of floating point operations needed, depending on the rate at which a 3-D compass is polled. In both cases specialization was very relevant, since a series of arithmetic operations (written separately for clarity) was reduced to just a couple of them. In the case of the highest compass polling rates, optimization of arithmetic operations (via unboxing of values) was of paramount importance and the resulting executable was only about 20% slower that a hand-written C counterpart. Other than that, compilation of control to low-level code was the second more important factor

in the speedup obtained.

References

1. P. A. Bigot and S. K. Debray. A simple approach to supporting untagged objects in dynamically typed languages. In *International Logic Programming Symposium*, pages 257–271, 1995.

2. J. Blauert. *Spatial Hearing : The Psychophysics of Human Sound Localization.* The MIT Press, 1983.

3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. P. (Eds.). The Ciao System. Reference Manual (v1.10). The Ciao System Documentation Series–TR CLIP3/97.1.10(04), School of Computer Science, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, August 2004. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

4. M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.

5. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *Eighth International Symposium on Functional and Logic Programming (FLOPS'06)*, Fuji Susono (Japan), April 2006.

6. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.

7. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

8. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

9. S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

10. X. Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albequerque, New Mexico, 1992.

11. M. McCarthy and H. Muller. No pingers: Ultrasonic indoor location sensing without rf synchonisation. Technical Report 003-004, University of Bristol, Department of Computer Science, May 2003.

12. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.

13. J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.

14. H. Muller and C. Randell. An event-driven sensor architecture for low power wearables. In *ICSE 2000, Workshop on Software Engineering for Wearable and Pervasive Computing*, pages 39–41. ACM/IEEE, June 2000.

15. J. Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 89–99. ACM Press, September 1989.

16. J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *PADL*, pages 91–105, 1999.

17. S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 636–666, Cambridge, Massachussets, USA, 26–28 August 1991. Springer-Verlag LNCS523.

18. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

19. C. Randell and H. L. Muller. The well mannered wearable computer. *Personal and Ubiquitous Computing*, 6(1):31–36, February 2002.

20. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

21. P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.

22. C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

23. G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report TR95-041, Department of Computer Science, University of North Carolina - Chapel Hill, November 1995.

24. M. Wolfe. How Compilers and Tools Differ for Embedded Systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM and IEEE Computer Society, September 2005. Keynote Speech.